Chipsmall Limited consists of a professional team with an average of over 10 year of expertise in the distribution of electronic components. Based in Hongkong, we have already established firm and mutual-benefit business relationships with customers from,Europe,America and south Asia,supplying obsolete and hard-to-find components to meet their specific needs.

With the principle of "Quality Parts,Customers Priority,Honest Operation,and Considerate Service",our business mainly focus on the distribution of electronic components. Line cards we deal with include Microchip,ALPS,ROHM,Xilinx,Pulse,ON,Everlight and Freescale. Main products comprise IC,Modules,Potentiometer,IC Socket,Relay,Connector.Our parts cover such applications as commercial,industrial, and automotives areas.

We are looking forward to setting up business relationship with you and hope to provide you with the best service and solution. Let us make a better world for our industry!

Contact us

Tel: +86-755-8981 8866 Fax: +86-755-8427 6832

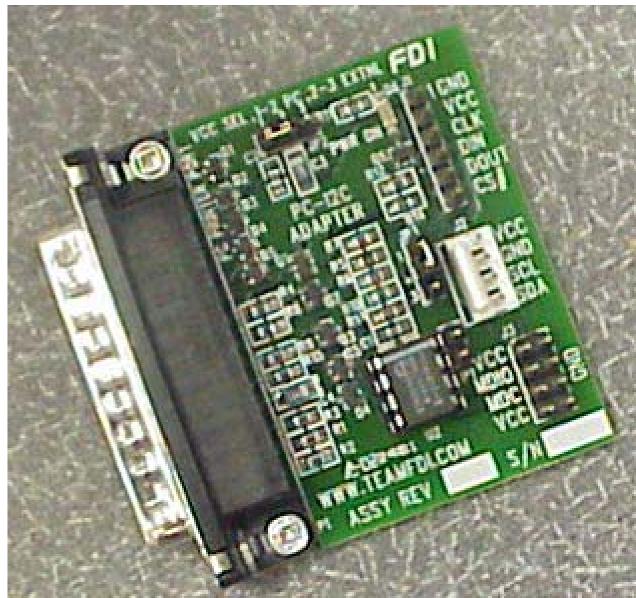
Email & Skype: info@chipsmall.com Web: www.chipsmall.com

Address: A1208, Overseas Decoration Building, #122 Zhenhua RD., Futian, Shenzhen, China

PC-I2C-DEV ©

with MDIO and SPI support

Software Developer User Manual



FDI *Future Designs, Inc.*
Your Development Partner

Information in this document is provided solely to enable the use of Future Designs, Inc. products. FDI assumes no liability whatsoever, including infringement of any patent or copyright. FDI reserves the right to make changes to these specifications at any time, without notice. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Future Designs, Inc. 2702 Triana Blvd SW, Huntsville, AL 35805-4074.

© 2004 Future Designs, Inc. All rights reserved.

Microsoft, MS-DOS, Windows, Microsoft Word are registered trademarks of Microsoft Corporation. Other brand names are trademarks or registered trademarks of their respective owners.

P:\PC PRoducts\PC-I2C\Docs\PC-I2C-DEV User Manual 3_E.doc, Revision 3.E, 3/11/2004 1:01 PM
Printed in the United States of America

Table of Contents

1. PC-I2C-DEV Overview 1
2. PC-I2C-DEV Installation 1
3. Using the PC-I2C-DEV in a VC++ Project 2
4. Three Example VC++ Projects 4
5. General Purpose Routines 5
   5.1 DriverAgentOpen 5
   5.2 DaOpenDevice 5
   5.3 DaCloseDevice 5
   5.4 DriverAgentClose 5
   5.5 SetupHardware 6
   5.6 DetectHardware 6
6. I2C Routines 7
   6.1 I2cReadSCL 7
   6.2 I2cDropSCL 7
   6.3 I2cRaiseSCL 7
   6.4 I2cReadSDA 7
   6.5 I2cDropSDA 7
   6.6 I2cRaiseSDA 8
   6.7 I2cGenerateStartCondition 8
   6.8 I2cGenerateRepeatedStartCondition 8
   6.9 I2cGenerateStopCondition 8
   6.10 I2cWriteByte 8
   6.11 I2cReadByte 9
   6.12 I2cWriteDevice 9
   6.13 I2cReadDevice 9
   6.14 I2cReadMemory 10
   6.15 I2cWriteMemory 10
   6.16 I2cReadMemory16 11
   6.17 I2cWriteMemory16 12
7. MDIO Routines 13
   7.1 MdioReadMDC 13
   7.2 MdioDropMDC 13
   7.3 MdioRaiseMDC 13
   7.4 MdioReadMDIO 13
   7.5 MdioDropMDIO 13
   7.6 MdioRaiseMDIO 14
   7.7 MdioSendBits 14
   7.8 MdioReadBits 14
   7.9 MdioSendPreamble 14
   7.10 Mdio22ReadWord 15
   7.11 Mdio22WriteWord 15
   7.12 Mdio45ReadWord 16
   7.13 Mdio45WriteWord 16

8. SPI Routines 17
8.1 SpiReadDIN 17
8.2 SpiDropDOUT 17
8.3 SpiRaiseDOUT 17
8.4 SpiReadCLK 17
8.5 SpiDropCLK 17
8.6 SpiRaiseCLK 18
8.7 SpiDropCS 18
8.8 SpiRaiseCS 18
8.9 SpiShiftReg 18
8.10 SpiClockWait 19
9. System Definitions 20
9.1 Function Return Values 20
9.2 nLoopsPerUsec Determination 20
9.3 NLoopsPerUsec Estimated Values 21

1. PC-I2C-DEV Overview

The PC-I2C-DEV Software Developer Kit contains all of the tools necessary to access and control the PC-I2C hardware from a custom application. The PC-I2C-DEV can be used with Visual C/C++ or any other programming language that supports the use of DLLs. (Note that the PC-I2C-DEV was generated and tested using Microsoft Visual C/C++ 6.0, but there is no reason that it will not work in all Microsoft Windows software development environments such as Borland C/C++, Microsoft Visual Studio, or any programming software which utilizes DLL.)

The PC-I2C-DEV consists of the following:

- A complete PC-I2C-KIT which includes
  - An FDI Installation and Support CDROM
  - PC-I2C Board with PCF8582 EEPROM
  - 4-pin connecting cable (18" length)
  - PC-I2C-KIT Quick Start Manual
  - Registration form for the PC-I2C parallel port adapter
- PC-I2C-DEV Software Developer Installation CDROM
  - Three user configurable DLL examples
- PC-I2C-DEV Software Developer User Manual

2. PC-I2C-DEV Installation

Insert the PC-I2C-DEV Installation CDROM into the proper drive. If the setup program does not automatically run, open the PC-I2C-DEV Installation CDROM from "My Computer" and run "setup.exe" by double-clicking on it. Follow the instructions on the subsequent screens. When the installation is completed, the computer will have to be re-booted in order for the installation to complete.

The Setup.exe program installs the files needed for the standard operation of the PC-I2C-KIT and those needed to add PC-I2C functionality to a custom application. The required .DLL and .SYS files will be copied to the appropriate system directories and the required modifications to the registry will be made. The remaining files will be copied into the following subdirectories (Assuming that the default directories were used during the installation procedure).

- C:\Program Files\FDI\PC-I2C – This directory contains the standard PC-I2C application and help files.
- C:\Program Files\FDI\PC-I2C\DDF – This directory contains the Device Descriptor Files (DDF) used by the standard PC-I2C-KIT software. See the PC-I2C software on-line help for details.
- C:\Program Files\FDI\PC-I2C\SDF – This directory contains the Sequence Descriptor Files (DDF) used by the standard PC-I2C-KIT software. See the PC-I2C software on-line help for details.

- C:\Program Files\FDI\PC-I2C\DEV – This directory contains the files that must be used by your custom application to access the PC-I2C hardware. This subdirectory also contains the electronic version of this user manual.
- C:\Program Files\FDI\PC-I2C \DLL_Example – This directory contains a Microsoft VC++ example project that can be used to verify that the PC-I2C hardware and software are functioning properly.
- C:\Program Files\FDI\PC-I2C \DTMF_Interface – This directory contains a Microsoft VC++ example project that will interface with any PCD3311C or PCD3312C compatible musical-tone generator.
- C:\Program Files\FDI\PC-I2C \LCD_Interface – This directory contains a Microsoft VC++ example project that will interface with any PCF21xxC family compatible LCD Driver.

3. Using the PC-I2C-DEV in a VC++ Project

In order to add the functionality of the PC-I2C-DEV to a custom application, you must perform the following steps. Note that you can use the example project located in the DLL_Example subdirectory as a reference for implementing these steps.

1. Make sure that the PC-I2C-DEV installation program was run from the installation disk. This program installs the software and makes the modifications to the registry that are needed for proper operation of the PC-I2C-DEV.

2. Copy the following files from the \DEV subdirectory (C:\Program Files\FDI\PC-I2C\DEV if the default installation location was used) to the directory that contains the source code for your custom application.
    a. Pport_Proxy.lib
    b. DriverAgent.lib
    c. Pport_Proxy.h
    d. DriverAgent.h
    e. Remap.h

3. In the VC++ environment, add the .lib files to your project by opening the “Project” menu, selecting “Add to Project”, and clicking on “Files”. Select the following files from the dialog box and click “OK”.
    a. Pport_Proxy.lib
    b. DriverAgent.lib

4. Add the following line to each of the files in your project that will access the PC-I2C software:
   #include “Pport_Proxy.h”

5. Create a global variable that will hold the handle of the PC-I2C device as follows:
   HCLIENTDEVICE g_hPCI2C = NULL;

6. During your application's initialization, add the following PC-I2C kernel initialization code:
```
DEVSTATUS devStatus = DEVSTATUS_SUCCESS ;
if ( API_SUCCESS(devStatus=DriverAgentOpen()) )
{
   // Open the "PPort" device
   devStatus=DaOpenDevice(_TEXT("PPort"), &g_hPCI2C, NULL);
}
```

7. Setup the PC-I2C hardware.
   SetupHardware(CPU_SPEED, PULSE_WIDTH, PORT_ADDR);

8. Call the PC-I2C-DEV routines as needed to perform the desired tasks.

9. When your application terminates, release the handle to the PC-I2C resources.
```
// Release the "PCI2C" handle if it exists
if ( g_hPCI2C ) DaCloseDevice(g_hPCI2C) ;
// Close the Kernel mode driver
DriverAgentClose() ;
```

4. Three Example VC++ Projects

The subdirectory “DLL_Example” contains an example project created with the Microsoft VC++ 6.0 MFC application wizard. The example project shows how to interface a custom application to the PC-I2C-DEV software to access the on-board PCF8582C I2C EEPROM.

The subdirectory “DTMF_Interface” contains an example project created with the Microsoft VC++ 6.0 MFC application wizard. The example project shows how to interface a custom application to the PC-I2C-DEV software to access a PCD3312P DTMF Generator.

The subdirectory “LCD_Interface” contains an example project created with the Microsoft VC++ 6.0 MFC application wizard. The example project shows how to interface a custom application to the PC-I2C-DEV software to access a PCF21xxC family LCD Driver.

5. General Purpose Routines

These routines are used to initialize, setup, and terminate the PC-I2C DLL functionality. They are usually called only once by the custom application.

5.1 DriverAgentOpen

Prototype: **DEVSTATUS DriverAgentOpen(void)**
Function: This routine is called to start the Kernel Mode driver that controls the parallel port. It should be called once during the custom application's initialization.
Parameters: None
Returns: None

5.2 DaOpenDevice

Prototype: **DEVSTATUS DaOpenDevice(**
PCSTR pszName,
HCLIENTDEVICE *phDevice,
PCLIENTDEVICEINFO pClientInfo
)
Function: This routine is called to get a handle to the Kernel Mode driver's parallel port resource. It should be called once during the custom application's initialization after calling DriverAgentOpen().
Parameters: pszName
phDevice
pClientInfo
Returns: None

5.3 DaCloseDevice

Prototype: **DEVSTATUS DaCloseDevice(HCLIENTDEVICE *hDevice)**
Function: This routine is called to release the handle to the Kernel Mode driver's parallel port resource. It should be called once during the custom application's shutdown procedure before calling DriverAgentClose().
Parameters: hDevice
Returns: None

5.4 DriverAgentClose

Prototype: **void DriverAgentClose(void)**
Function: This routine is called to terminate the Kernel Mode driver that controls the parallel port. It should be called once during the custom application's shutdown procedure.

Parameters: None
Returns: None

5.5 SetupHardware

Prototype: **int SetupHardware(int nLoopsPerUsec, int n50Percent, int nPort)**

Function: This routine is called to set up the parameters to be used by the PC-I2C hardware. It should be called once during the custom application's initialization. The parameters nLoopsPerUsec and n50Percent are used to determine the effective bus speed. The value of nLoopsPerUsec can be calculated using the code in the example or can be estimated from the chart that is provided later in this manual.

Parameters: **nLoopsPerUsec** – Number of loops the processor executes in a uSec.
n50Percent – Fifty times the desired clock period in uSec. For example, a 100kHz clock would yield 500 (50*1/100,000)
nPort – The hardware address of the Parallel Port that is being used by PC-I2C (e.g. 0x378, 0x3bc, etc.)

Returns: Always returns a 0x00

5.6 DetectHardware

Prototype: **int DetectHardware(void)**

Function: This routine is called to detect the presence of the PC-I2C hardware on the parallel port. This routine can be called at any time.

Parameters: None

Returns: Returns TRUE if the board was detected and FALSE if it was not detected.

6. I2C Routines

These routines are used to support the I2C bus protocol. There are routines included to perform low level bit manipulation as well as routines to send entire messages across the I2C bus.

6.1 I2cReadSCL

Prototype: **int I2cReadSCL(void)**
Function: This routine returns the current state of the SCL line of the I2C interface.
Parameters: None
Returns: “0” if SCL is low, “1” if SCL is high

6.2 I2cDropSCL

Prototype: **int I2cDropSCL(void)**
Function: This routine forces the SCL line of the I2C interface low.
Parameters: None
Returns: Always returns I2C_NO_ERROR

6.3 I2cRaiseSCL

Prototype: **int I2cRaiseSCL(void)**
Function: This routine forces the SCL line of the I2C interface high.
Parameters: None
Returns: Always returns I2C_NO_ERROR

6.4 I2cReadSDA

Prototype: **int I2cReadSDA(void)**
Function: This routine returns the current state of the SDA line of the I2C interface.
Parameters: None
Returns: “0” if SDA is low, “1” if SDA is high

6.5 I2cDropSDA

Prototype: **int I2cDropSDA(void)**
Function: This routine forces the SDA line of the I2C interface low.
Parameters: None
Returns: Always returns I2C_NO_ERROR

6.6 I2cRaiseSDA

Prototype: **int I2cRaiseSDA(void)**
Function: This routine forces the SDA line of the I2C interface high.
Parameters: None
Returns: Always returns I2C_NO_ERROR

6.7 I2cGenerateStartCondition

Prototype: **int I2cGenerateStartCondition(void)**
Function: This routine generates a start condition on the I2C interface.
Parameters: None
Returns: Always returns I2C_NO_ERROR

6.8 I2cGenerateRepeatedStartCondition

Prototype: **int I2cGenerateRepeatedStartCondition(void)**
Function: This routine generates a repeated start condition on the I2C interface.
Parameters: None
Returns: Always returns I2C_NO_ERROR

6.9 I2cGenerateStopCondition

Prototype: **int I2cGenerateStopCondition(void)**
Function: This routine generates a stop condition on the I2C interface.
Parameters: None
Returns: Always returns I2C_NO_ERROR

6.10 I2cWriteByte

Prototype: **int I2cWriteByte(int nByte)**
Function: This function transmits a single byte to the I2C bus. It assumes that the bus is available, that the proper Start Condition has previously been generated, and that the slave device has been properly addressed.
Parameters: **nByte** – The byte to write to the I2C interface.
Returns: Returns I2C_NO_ACK if the slave device does not acknowledge the byte. Otherwise it returns I2C_NO_ERROR.

6.11 I2cReadByte

Prototype: **int I2cReadByte(int *nByte, int nLast)**

Function: This function reads a single byte from the I2C bus. It assumes that the bus is available, that the proper Start Condition has previously been generated, and that the slave device has been properly addressed. If the parameter nLast is FALSE, an ACK is generated after the byte is transmitted. Otherwise, no ACK is generated. The result of the read is saved in *nByte.

Parameters: ***nByte** – This is a pointer to the location that receives the byte read from the I2C bus.
nLast – This parameter determines if an ACK should be generated after the byte is transmitted. If nLast is FALSE, an ACK is generated. If nLast is TRUE, no ACK is generated.

Returns: Always returns I2C_NO_ACK. Also updates the value pointed to by nByte with the byte read from the I2C bus.

6.12 I2cWriteDevice

Prototype: **int I2cWriteDevice(int nDeviceAddress,
int nCount,
int nBuffer[],
int nRegWidth = 1)**

Function: This function is used to write a complete message to the I2C bus. It handles generation of the Start and Stop Conditions as well as properly addressing the Slave device.

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.
nCount – The number of words to write to the slave device.
nBuffer[] – A buffer that contains the bytes to write to the slave device.
nRegWidth – The width of each register and thus each word.

Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge any of the bytes that are transmitted. Otherwise returns I2C_NO_ERROR.

6.13 I2cReadDevice

Prototype: **int I2cReadDevice(int nDeviceAddress,
int nCount,
int nBuffer[],
int nRegWidth = 1)**

Function: This function is used to read a complete message from the I2C bus. It handles generation of the Start and Stop Conditions as well as properly addressing the Slave device. It also generates an ACK for every byte transmitted except for the final one. (This is a common method of terminating a read process on the I2C bus.)

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.
nCount – The number of words to read from the slave device.
nBuffer[] – A buffer that receives the bytes read from the slave device.
nRegWidth – The width of each register and thus each word.

Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge its address. Otherwise returns I2C_NO_ERROR. It also updates the contents of nBuffer with the read results.

6.14 I2cReadMemory

Prototype: **int I2cReadMemory(**
int nDeviceAddress,
int nMemoryAddress,
int nCount,
int nBuffer[],
int nRegWidth = 1
)

Function: This function reads a block of memory from an I2C memory device using 11-bit internal addressing. It handles the generation of the Start and Stop Conditions as well as properly addressing the Slave device. It also handles setting up the proper address to read and the proper ACK sequence for the read procedure.

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.
nMemoryAddress – The address within the slave device to begin reading.
nCount – The number of words to read from the slave device. (A maximum of 0x100 bytes can be read at a time.)
nBuffer[] – A buffer that receives the bytes read from the slave device.
nRegWidth – The width of each register and thus each word.

Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge its address or the memory address to read. Returns I2C_COUNT_TOO_BIG if a number larger than 0x100 is read in nCount. Otherwise returns I2C_NO_ERROR. It also updates the contents of nBuffer with the read results.

6.15 I2cWriteMemory

Prototype: **int I2cWriteMemory(**
int nDeviceAddress,
int nMemoryAddress,
int nCount,
int nBuffer[],
int nRegWidth = 1
)

Function: This function writes a block of memory to an I2C memory device using 11-bit internal addressing. It handles the generation of the Start and Stop Conditions as well as properly addressing the Slave device. It also handles setting up the proper address to write.

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.
nMemoryAddress – The address within the slave device to begin writing.
nCount – The number of words to write to the slave device. (A maximum of 0x10 bytes can be written at a time.)
nBuffer[] – A buffer that contains the bytes to write to the slave device.
nRegWidth – The width of each register and thus each word.

Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge any of the bytes written to it. Returns I2C_COUNT_TOO_BIG if a number larger than 0x10 is read in nCount. Otherwise returns I2C_NO_ERROR.

6.16 I2cReadMemory16

Prototype:

```
int I2cReadMemory16(
        int nDeviceAddress,
        int nMemoryAddress,
        int nCount,
        int nBuffer[],
        int nRegWidth = 1
        )
```

Function: This function reads a block of memory from an I2C memory device using 19-bit internal addressing. It handles the generation of the Start and Stop Conditions as well as properly addressing the Slave device. It also handles setting up the proper address to read and the proper ACK sequence for the read procedure.

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.
nMemoryAddress – The address within the slave device to begin reading.
nCount – The number of words to read from the slave device. (A maximum of 0x10000 bytes can be read at a time.)
nBuffer[] – A buffer that receives the bytes read from the slave device.
nRegWidth – The width of each register and thus each word.

Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge its address or the memory address to read. Returns I2C_COUNT_TOO_BIG if a number larger than 0x10000 is read in nCount. Otherwise returns I2C_NO_ERROR. It also updates the contents of nBuffer with the read results.

6.17 I2cWriteMemory16

Prototype: **int I2cWriteMemory16(**
int nDeviceAddress,
int nMemoryAddress,
int nCount,
int nBuffer[],
int nRegWidth = 1
)

Function: This function writes a block of memory to an I2C memory device using 19-bit internal addressing. It handles the generation of the Start and Stop Conditions as well as properly addressing the Slave device. It also handles setting up the proper address to write.

Parameters: **nDeviceAddress** – The I2C bus address of the slave device.
nMemoryAddress – The address within the slave device to begin writing.
nCount – The number of words to write to the slave device. (A maximum of 0x10 bytes can be written at a time.)
nBuffer[] – A buffer that contains the bytes to write to the slave device.
nRegWidth – The width of each register and thus each word.

Returns: Returns I2C_NO_ACK if the slave device fails to acknowledge any of the bytes written to it. Returns I2C_COUNT_TOO_BIG if a number larger than 0x10 is read in nCount. Otherwise returns I2C_NO_ERROR.

7. MDIO Routines

7.1 MdioReadMDC

Prototype: **int MdioReadMDC(void)**
Function: This routine returns the current state of the MDC line of the MDIO interface.
Parameters: None
Returns: "0" if MDC is low, "1" if MDC is high

7.2 MdioDropMDC

Prototype: **int MdioDropMDC(void)**
Function: This routine forces the MDC line of the MDIO interface low.
Parameters: None
Returns: Always returns I2C_NO_ERROR

7.3 MdioRaiseMDC

Prototype: **int MdioRaiseMDC(void)**
Function: This routine forces the MDC line of the MDIO interface high.
Parameters: None
Returns: Always returns I2C_NO_ERROR

7.4 MdioReadMDIO

Prototype: **int MdioReadMDIO(void)**
Function: This routine returns the current state of the MDIO line of the MDIO interface.
Parameters: None
Returns: "0" if MDIO is low, "1" if MDIO is high

7.5 MdioDropMDIO

Prototype: **int MdioDropMDIO(void)**
Function: This routine forces the MDIO line of the MDIO interface low.
Parameters: None
Returns: Always returns I2C_NO_ERROR

7.6 MdioRaiseMDIO

Prototype: **int MdioRaiseMDIO(void)**
Function: This routine forces the MDIO line of the MDIO interface high.
Parameters: None
Returns: Always returns I2C_NO_ERROR

7.7 MdioSendBits

Prototype: **int MdioSendBits(**
int nData,
int nCount
)
Function: This routine transfers any number of bits (up to 32) on the MDIO interface by placing the bit on the bus and transitioning the clock.
Parameters: **nData** – The data pattern to transmit. Bits are transmitted starting with the least significant bit.
nCount – Number of bits to transmit.
Returns: Always returns I2C_NO_ERROR

7.8 MdioReadBits

Prototype: **int MdioReadBits(**
int *nData,
int nCount
)
Function: This routine reads any number of bits (up to 32) from the MDIO interface.
Parameters: **nData** – Value that receives the result of the read operation.
nCount – Number of bits to read.
Returns: Always returns I2C_NO_ERROR. Also updates the value of nData with the bits read from the bus.

7.9 MdioSendPreamble

Prototype: **int MdioSendPreamble(void)**
Function: This routine transmits 32 high bits on the MDIO interface to create the preamble that is required by the MDIO specification.
Parameters: None
Returns: Always returns I2C_NO_ERROR.

7.10 Mdio22ReadWord

Prototype: **int Mdio22ReadWord(**
int nPhyAddr,
int nRegAddr,
int *nData
)

Function: This routine reads a single 32bit word from the MDIO (Clause 22) interface. The resultant word is returned in the variable nData.

Parameters: **nPhyAddr** – This value is the address of the slave device on the MDIO bus.
nRegAddr – This is the address of the register within the slave device that is to be read.
nData – This is the location that will receive the results of the read operation.

Returns: Always returns I2C_NO_ERROR. Also updates the value of nData with the value read from the bus.

7.11 Mdio22WriteWord

Prototype: **int Mdio22WriteWord(**
int nPhyAddr,
int nRegAddr,
int nData
)

Function: This routine writes a single 32bit word to the MDIO (Clause 22) interface.

Parameters: **nPhyAddr** – This value is the address of the slave device on the MDIO bus.
nRegAddr – This is the address of the register within the slave device that is to be written.
nData – This is the data that will be written to the slave device.

Returns: Always returns I2C_NO_ERROR.

7.12 Mdio45ReadWord

Prototype: **int Mdio45ReadWord(**
int nPortAddr,
int nDevAddr,
int nRegAddr,
int *nData
)

Function: This routine reads a single 32bit word from the MDIO (Clause 45) interface. The resultant word is returned in the variable nData.

Parameters: **nPortAddr** – This value is the address of the slave device on the MDIO bus.
nDevAddr – This is the device page address of the slave device.
nRegAddr – This is the address of the register within the slave device that is to be read.
nData – This is the location that will receive the results of the read operation.

Returns: Always returns I2C_NO_ERROR. Also updates the value of nData with the value read from the bus.

7.13 Mdio45WriteWord

Prototype: **int Mdio45WriteWord(**
int nPortAddr,
int nDevAddr,
int nRegAddr,
int nData
)

Function: This routine writes a single 32bit word to the MDIO (Clause 24) interface.

Parameters: **nPortAddr** – This value is the address of the slave device on the MDIO bus.
nDevAddr – This is the device page address of the slave device.
nRegAddr – This is the address of the register within the slave device that is to be written.
nData – This is the data that will be written to the slave device.

Returns: Always returns I2C_NO_ERROR.

8. SPI Routines

8.1 SpiReadDIN

Prototype: **int SpiReadDIN(void)**
Function: This routine returns the current state of the DIN line of the SPI interface.
Parameters: None
Returns: "0" if DIN is low, "1" if DIN is high

8.2 SpiDropDOUT

Prototype: **int SpiDropDOUT(void)**
Function: This routine forces the DOUT line of the SPI interface low.
Parameters: None
Returns: Always returns I2C_NO_ERROR

8.3 SpiRaiseDOUT

Prototype: **int SpiRaiseDOUT(void)**
Function: This routine forces the DOUT line of the SPI interface high.
Parameters: None
Returns: Always returns I2C_NO_ERROR

8.4 SpiReadCLK

Prototype: **int SpiReadCLK(void)**
Function: This routine returns the current state of the CLK line of the SPI interface.
Parameters: None
Returns: "0" if CLK is low, "1" if CLK is high

8.5 SpiDropCLK

Prototype: **int SpiDropCLK(void)**
Function: This routine forces the CLK line of the SPI interface low.
Parameters: None
Returns: Always returns I2C_NO_ERROR

8.6 SpiRaiseCLK

Prototype: **int SpiRaiseCLK(void)**
Function: This routine forces the CLK line of the SPI interface high.
Parameters: None
Returns: Always returns I2C_NO_ERROR

8.7 SpiDropCS

Prototype: **int SpiDropCS(void)**
Function: This routine forces the CS line of the SPI interface low.
Parameters: None
Returns: Always returns I2C_NO_ERROR

8.8 SpiRaiseCS

Prototype: **int SpiRaiseCS(void)**
Function: This routine forces the CS line of the SPI interface high.
Parameters: None
Returns: Always returns I2C_NO_ERROR

8.9 SpiShiftReg

Prototype: **int SpiShiftReg(**
int OutGoing[],
int WordWidth,
int InComing[],
int UseCS
)

Function: This routine takes the value from OutGoing[] and shifts it out on DOUT while storing the value shifted in from DIN to Incoming[].

Parameters: **int OutGoing[]** – This value is the command to be shifted out to the target device.
int WordWidth – This value is the width of the command in bits.
int InComing[] – This value is the data shifted in from DIN during transmission of the command.
int UseCS – This value is the polarity of the CS line during transmission. “0” signifies that CS should be high during transmission. “1” signifies that CS should be low during transmission. “2” signifies that CS is not applicable.

Returns: Always returns I2C_NO_ERROR. The information shifted in on DIN is stored in InComing[].

8.10 SpiClockWait

Prototype: **int SpiClockWait(**
BOOL WaitForValue,
int WaitClocks,
int UseCS
)

Function: This routine forces the CS line of the SPI interface high. The CS line is held in this state until a logic value matching WaitForValue is read on DIN. When this is read, or SpiClockWait has waited an amount of time equal to WaitClocks number of clocks, the function will return I2C_NO_ERROR.

Parameters: **BOOL WaitForValue** – The logic value on the DIN line which specifies that SpiClockWait should return.
int WaitClocks – The number of clocks to count before SpiClockWait should give up and return regardless of the value on the DIN line.
int UseCS – This value is the polarity of the CS line during transmission. “0” signifies that CS should be high during transmission. “1” signifies that CS should be low during transmission. “2” signifies that CS is not applicable.

Returns: Always returns I2C_NO_ERROR

9. System Definitions

This section contains miscellaneous system definitions and notes for using the PC-I2C-DEV.

9.1 Function Return Values

The following values can be returned by the Developer Kit routines. They are returned by the SPI and MDIO routines as well as the I2C routines. These values are declared in the Pport_Proxy.h file.

- **I2C_NO_ERROR** – Functions return this value if no error occurs. This is the standard return value.
- **I2C_NO_ACK** – Functions return this value when the slave device fails to acknowledge a byte that was transferred over the bus.
- **I2C_COUNT_TOO_BIG** – Functions return this value when the amount of data to move is too large for the internal buffers.

9.2 nLoopsPerUsec Determination

The value of the parameter nLoopsPerUsec, which is used by the SetupHardware() routine, is used to specify the processing speed of the host computer. This value can be calculated directly, using the code snippet below, or can be estimated from values given in the table.

```
//
// Determine nLoopsPerUsec value
//
LARGE_INTEGER Freq, Start, Finish;
_int64 HPAverage = 0;
_int64 HPDiff = 0;
_int64 HPusec = 0;

int nCount = 0;
QueryPerformanceFrequency( &Freq);

// Loop 10 times
for (int j=0; j<10; j++)
{
      QueryPerformanceCounter( &Start);
      for (int i=0;i<0x00800000;i++) nCount += i;
      QueryPerformanceCounter( &Finish);
      HPDiff = Finish.QuadPart-Start.QuadPart;
      HPAverage += (HPDiff/10);
}
HPusec = ((Freq.QuadPart/1000)*0x20ce)/HPAverage;
```

9.3 NLoopsPerUsec Estimated Values

The following table gives some estimated values for the parameter nLoopsPerUsec, which is used by the SetupHardware() routine. The values will give adequate results for most applications. If more precise timing is required, use the code snippet above to calculate the value at run time.

Processor Type	nLoopsPerUsec
Pentium 400 MHz	100
Pentium 700 MHz	220
Pentium 1.1 GHz	275
Athlon 1.2 GHz	305